IMPROVING REAL-WORLD APPLICATION PERFORMANCE UTILIZING PARALLELIZED COMMODITY COMPONENTS

Version 1
Submitted to
Software: Practice and Experience
August 20, 2003

Michael Davis
Randy Smith*
Brandon Dixon
Allen Parrish
David Cordes

Department of Computer Science
The University of Alabama
Box 870290
Tuscaloosa, Alabama 35487-0290

August 2003

Key Words: Parallel computing, commodity components, performance analysis

Abstract

Commodity computing hardware continues to increase performance while decreasing price. This combination is driving a renewed interest in parallel and distributed computing. In this study, we examine the performance of an existing application in a ten-node computing cluster using commodity off-the-shelf components. The application is a statistical analysis software package that processes categorical data used by state public safety programs. The study examines various network topologies and focuses on minimizing the modifications required to distribute the application. We conclude that parallel computing using commodity components is an effective mechanism to increase the performance of real-world applications especially when the underlying application architectures are flexible enough to allow for efficient reuse of the existing code.

1 INTRODUCTION

The combination of rapidly rising performance and falling costs for commodity off-the-shelf (COTS) computing components has caused renewed interest in the field of parallel and distributed computing. Supercomputers based on COTS components provide performance that is competitive with more traditional, proprietary systems for a fraction of the cost. For example, in 1997, the Loki system at Los Alamos National Laboratory yielded a cost-performance ratio that was more than three times as the values realized by a number of proprietary supercomputing machines available at the time [7]. Recently, the Galaxy system at Stony Brook University delivered a cost-performance ratio nearly an order of magnitude better than that offered by proprietary systems [1]. A long list of such projects exist, most of which have their roots in the Beowulf project initiated at the Center of Excellence in Space Data and Information Sciences (CESDIS) in 1994 [2]. Clusters that use off-the-shelf components can now be found among the

* Corresponding author

top 500 supercomputers in the world, including the MCR Linux cluster at Lawrence Livermore National Laboratories. This cluster is currently ranked fifth with a benchmark score of 5.6 trillion floating point operations per second [6].

In this paper, we examine performance gains using a computing cluster constructed from COTS components. The cluster is tested using an existing application, modified to support distributed systems. A key factor in this work is minimizing the modifications necessary to produce an effective distributed version of the software. Thus, the application's modifications reuse existing system components whenever possible. The effects of this design decision on both ease of development and performance of the system are examined to determine the minimal amount of modification necessary to achieve our desired performance gains.

The following section describes the application and the data domain. Section 3 discusses the study configuration including hardware, application and data distribution. Section 4 overviews the experimental methodology. Section 5 presents the results of this study. Our conclusions and avenues for future work are presented in Section 6.

2 APPLICATION OVERVIEW

The application studied is the Critical Analysis and Reporting Environment (CARE). CARE is a software system developed for performing statistical analyses on static sets of categorical data. CARE was originally designed to analyze automobile crash data; however, most of its techniques are applicable to any type of categorical data [4,5].

CARE is similar to a relational database management system (RDBMS). Specifically, it is designed to work with large amounts of data in a tabular format, can limit analysis to a subset of the records based on individual column values, and possesses some limited facilities for simulating joins between related tables. However, since CARE is designed to perform statistical analysis of static categorical data rather than general-purpose queries, many of the capabilities required by an RDBMS are omitted from CARE.

CARE maximizes the efficiency of performing statistical analysis on subsets of records and subsets of attributes. CARE routinely outperforms commercial statistics software packages (such as SPSS) for the types of analysis that it performs [5]. Much of CARE's performance comes from its data storage method, which is optimized to support the most frequent analyses performed by its users [5].

2.1 CARE Data

CARE manipulates data and metadata in a manner that is roughly analogous to a single table in a relational database system. Typically, a CARE dataset represents a set of data collected over a fixed period of time, such as one calendar year. This data may be combined to create larger data sets. CARE datasets are generated offline using a combination of both general-purpose and custom-designed utilities. These utilities gather data from the original data source (usually a relational database or a flat file), convert it into categorical data according to specific user-defined rules, and write the data to the various files that make up a CARE dataset.

CARE employs a column-major paradigm for representing the records in a CARE dataset. Each column of data represents a unique attribute for the dataset's records. In CARE, each column is stored in a separate file. This structure facilitates efficient generation of basic statistical information (sums, means, and frequencies) for small collections of attributes [5]. Since the ordering across all the attribute files is consistent, CARE can reassemble an individual record by combining the values at a given location from each attribute file. Figure 1 gives an example where attribute 1 is county, attribute 2 is day of week and attribute M is month. Recall that each column is physically stored in a separate file.

	File 1 (Attribute 1)	File 2 (Attribute 2)	… …	File M (Attribute M)
Record 0	12	0		12
Record 1	2	6		10
… …				
Record N	23	4		7

Figure 1: CARE Record and Attribute Storage

CARE allows its users to perform analyses on subsets of the dataset through the use of *filters*. Filters are user-generated expressions that contain comparison clauses (comparing the value of an attribute for a particular record with some constant value) connected by the Boolean algebraic operators AND, OR, and NOT. Sample filters include "Day-of-week = Saturday" and "County = Washington AND Month = December." The CARE system uses these filters to construct a Boolean array (of length N) that determines whether or not a given record should be included in or excluded from the subset. True values in this array indicate records that are included in the filter's subset, while false values indicate records that are rejected by the filter. The filter's resultant Boolean array can be efficiently saved to disk with eight Boolean values packed per byte.

CARE supports several different descriptive statistical analyses for categorical data. The simplest of these analyses is a frequency distribution. In this paper, our experiments were conducted using frequency distributions.

2.2 CARE Architecture

The current CARE application is composed of a set of Component Object Model (COM) objects contained within a single dynamic-link library (DLL) file named "CARE.DLL." Client applications use these COM objects to perform analysis techniques on the available datasets. The client applications are responsible for displaying the results to the user. The two primary client applications that use the CARE.DLL server component are a graphical user interface client developed in Microsoft Visual Basic 6.0 that is intended for local use on a workstation and a web version that performs the calculations on a web server and displays the results using Active Server Pages (ASP). Figure 2 gives an overview of the current architecture.

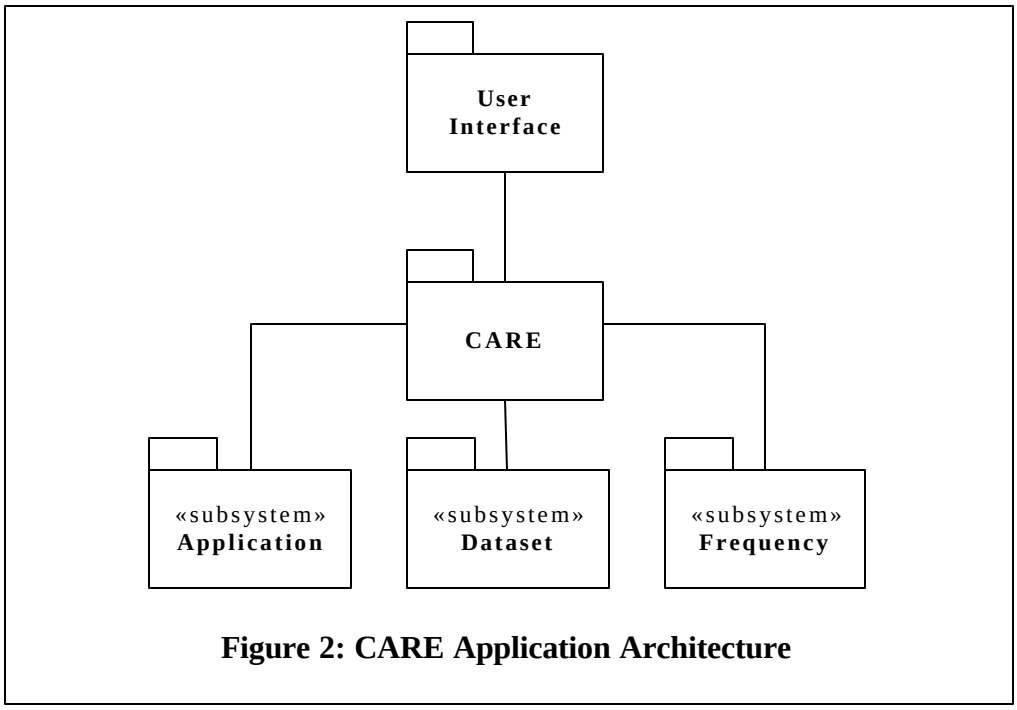


Figure 2: CARE Application Architecture

It is this architecture that forms the basis for developing the three distinct distributed versions of CARE used in the study.

3 STUDY ENVIRONMENT

As mentioned previously, this research examines performance characteristics of parallel computing using commodity components. Before discussing the results of the study, we first overview the hardware utilized, and the distribution of the algorithm to this environment. Section

3.1 describes our hardware configuration. Section 3.2 describes how the application and the data were distributed across this hardware.

3.1 Hardware Configuration

We constructed a ten-node cluster using off-the-shelf components. Each node in the cluster contains the following hardware:

- ABIT TH7II-RAID motherboard
- 1.7 GHz Intel Pentium 4 central processing unit (CPU)
- 512 GB of PC800 RDRAM
- ATI Radeon 7000 graphics card
- Intel PRO/100+ Management 10/100 Ethernet adapter
- 2 IBM Deskstar 120GXP 80 GB hard drives (configured as a RAID-0 array)

Each node runs Windows XP Professional as its operating system. The cluster utilizes a Cisco Catalyst 2950 24-port 10/100 Ethernet switch. A keyboard-video-mouse (KVM) switch is used to allow access to each node. The total cost of all the hardware for the cluster (also including rack-mount cases) was approximately $10,000 at the time of purchase (Spring 2002). Hardware problems caused one of the nodes to become inoperable during testing. Therefore, the results presented here were generated using only nine of the available ten nodes.

3.2 Application Configuration

Three unique versions of a distributed CARE system were developed. Each of these three versions is described in the following paragraphs. In the remainder of the paper, we refer to these as simply Versions 1, 2 and 3.

Version 1 is based on the architecture of the current version of the CARE system. The distributed version of the current application relies on the Distributed Component Object Model (DCOM) to communicate between processors. Versions 2 and 3 are based on a new version of CARE currently being developed using Microsoft's .NET Framework. The distributed variations of this version use the .NET Remoting facility to communicate between processors.

In Version 1, we maintained the existing CARE architecture and utilized the Distributed Component Object Model (DCOM) to distribute the processing. This task was straightforward. We added an intermediate layer that acts as a 'middleman' between the user interface and the existing components. This approach, depicted in Figure 3, required only a trivial modification to the User Interface component (one line was changed).

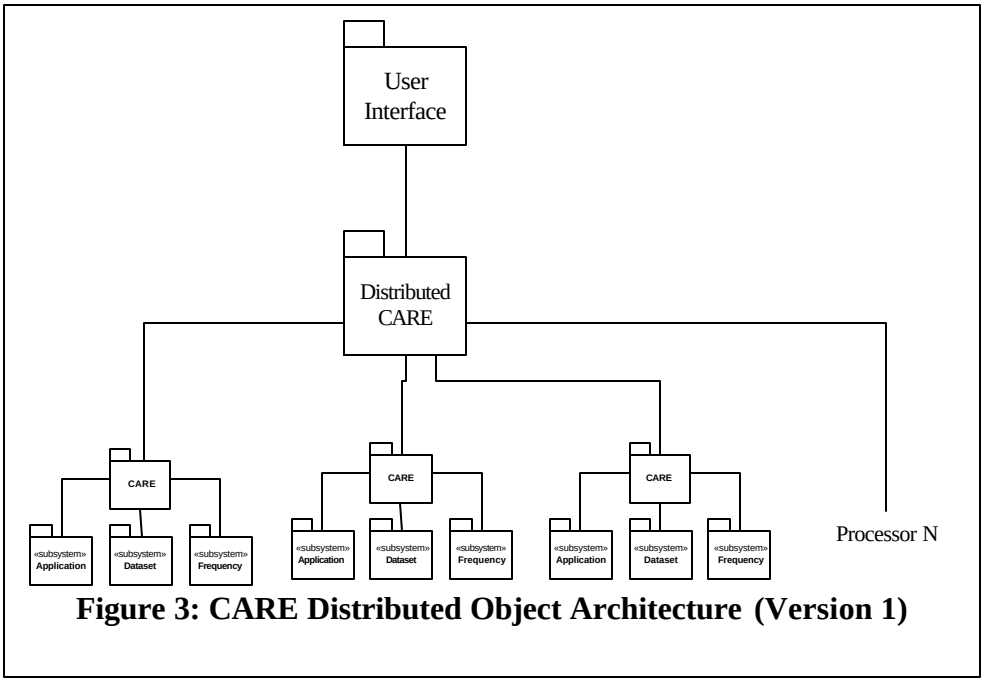


Figure 3: CARE Distributed Object Architecture (Version 1)

The resultant architecture is a classic distributed star configuration. As shown by Figure 3, our Distributed CARE component sends a single request to each node for processing and then aggregates the results. If more than one attribute is required by the query, the Distributed CARE component waits for the first attribute to be completely processed and then sends the next attribute to each node for processing.

Versions 2 and 3 of the CARE architecture rely on a "plug-in" architecture that is based on Microsoft .NET technology. This architecture has been developed as an alterative to the existing CARE architecture. Services, such as frequency or cross-tabulations, are provided to the CARE application via a set of plug-ins. Distributed versions of this architecture simply require the creation of new plug-ins for the application. This flexibility inherent in this approach permits the development of two different treatments in terms of task assignment and data aggregation.

Version 2, depicted in Figure 4, uses a generalized tree structure. In this model, each leaf is a processor performing calculations. Selected nodes then merge the resulting data and report back to the CARE application. While this approach still requires the root node to distribute individual attribute requests, the approach reduces the overhead on the CARE application.

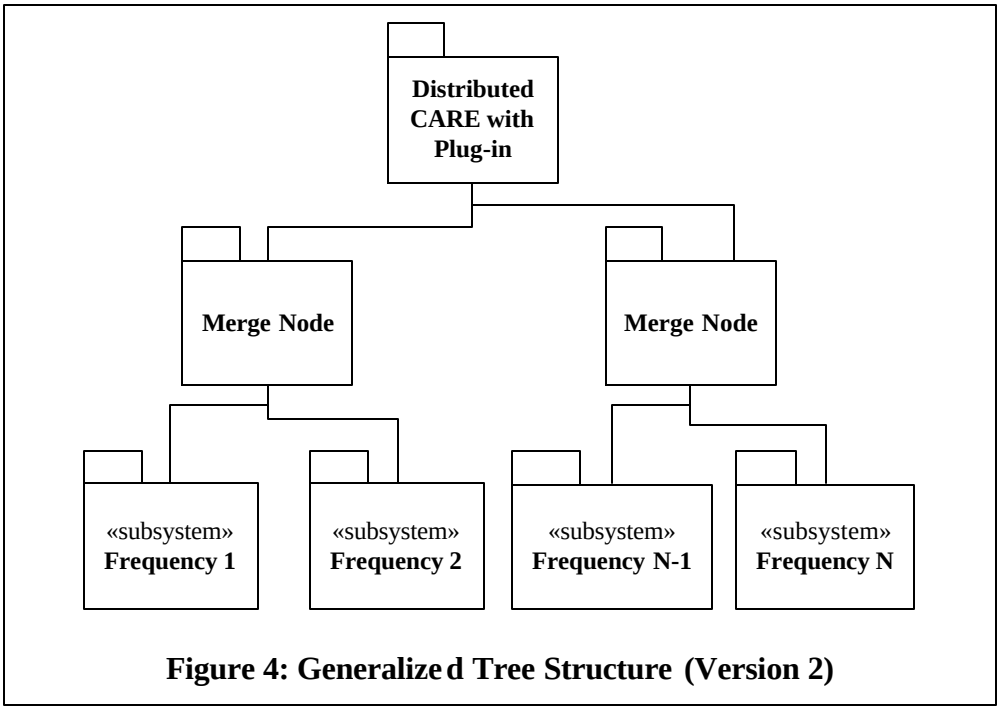


Figure 4: Generalized Tree Structure (Version 2)

Version 3 utilizes the flexibility of the "plug-in" architecture to provide asynchronous communications to the nodes. This architecture, depicted in Figure 5, is identical to the classical star structure shown above except that the central node distributes all requests in batch mode to each node. The computation nodes, such as the Frequency Node shown below, report attribute results back as they are calculated.

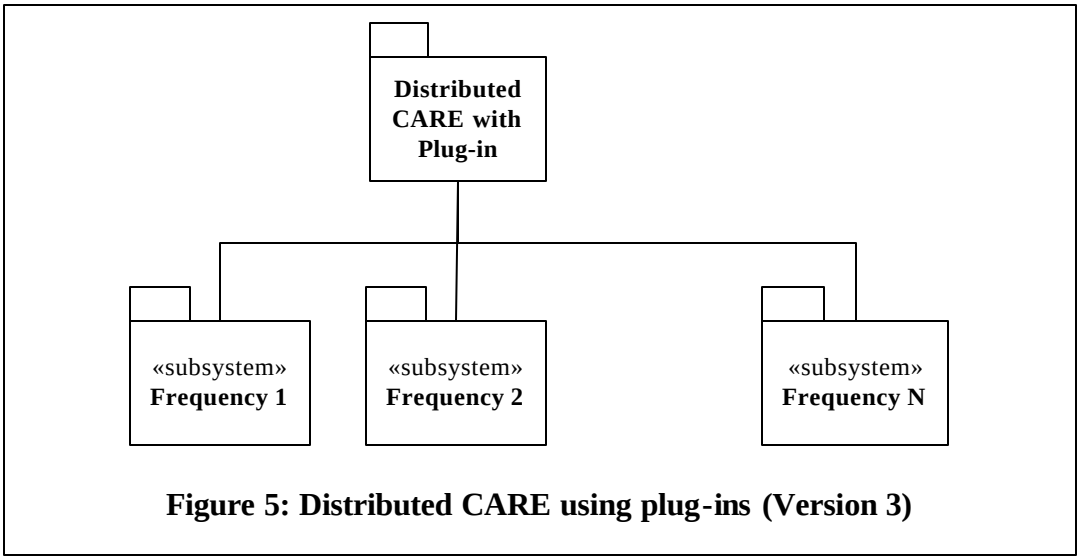


Figure 5: Distributed CARE using plug-ins (Version 3)

3.3 Data Configuration

The final step in converting our application to a distributed application is to distribute the data to each processor. To distribute a CARE dataset, each processor is given a subset of the original dataset's records. The simplest method of doing this is to assign records to processors in large contiguous blocks. That is, creating the distributed datasets is accomplished by splitting the attributes and filter files. Figure 6 illustrates this approach.

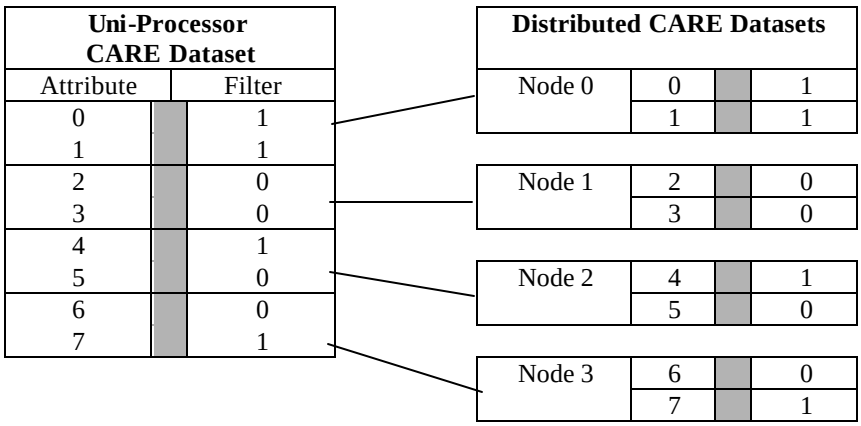


Figure 6: Distribution of a CARE Dataset Across Multiple Processors

However, this method could result in clustering of the data, which might unbalance the workload of the individual nodes in the system. Thus, we instead employ a random distribution of records among the nodes. The number of records assigned to each processor is arbitrary, although for a system composed of homogeneous processors, assigning an equal number of records to each processor is likely to produce the best results.

4 EXPERIMENTAL METHODOLOGY

The structure of the CARE system suggests that significant performance benefits could result from distributing the computations across multiple processors. Furthermore, CARE datasets can easily be divided between the nodes of a distributed system. The basic analysis techniques supported by CARE (such as frequency distributions) do not require communication between nodes to perform the actual computation. Communication between nodes is only necessary when transmitting calculation requests and returning calculation results. Furthermore, several of the existing components can be reused in the distributed versions, greatly reducing development time.

Since CARE datasets typically contain static data rather than dynamic data, the problem of updating records in a distributed RDBMS, which would normally require an expensive two-phase commit protocol to maintain the property of atomicity [3], can be ignored. Additionally, the problem of joining multiple tables in a distributed database [3] is irrelevant to CARE. CARE has

no direct join function, and the nearest equivalent, the filter conversion function, can be computed once on the single-processor forms of the datasets being joined. The resulting filter is then split and distributed among the processors as described below.

In theory, given a distributed CARE system running on an M-node cluster analyzing a dataset containing N records, an O(N) operation (such as frequency analysis) would only take O(N/M) time. However, a number of factors may affect the system's actual running time, causing it to be faster or slower than the expected running time. First, transmission of requests and results across the network adds transmission delay overhead. Second, since most of CARE's analysis techniques contain percentage data, the root node of the system must calculate the percentage data after merging the results from all the other nodes. Since this computation requires that the total number of records being analyzed is known, the root node cannot compute this any earlier. This delays the delivery of the final results to the user. Third, when the existing calculation components are reused in a distributed system, this percentage data is (unnecessarily) calculated for the subset of the data stored at each node. Since the percentages are only meaningful for that subset, they are discarded when the results are sent back to the initiating process. Removing this calculation on the individual nodes could reduce the overhead. However, since the overhead added by these three factors remains small when compared to the performance gains that can be realized by parallelizing the calculations, the results were anticipated to be very close to the theoretical improvement in performance.

Finally, we note that other factors may actually work to increase the speed beyond the expected linear increase. By reducing the number of records each node must examine, the likelihood of being able to cache large amounts of the data in main memory is increased. If a nearly linear speed increase is achieved by distributing the data, the improved cache hit rate caused by reducing the node's data could easily improve performance beyond the expected linear increase.

4.1 Software and Test Data Configuration

All three versions of the application were installed on each of the 8 nodes used in the performance analysis. To allow for automated testing, the normal CARE graphical user interface was not used. Instead, command-line equivalents were developed for each version that executed a user-specified frequency distribution calculation. The command-line clients measure the total time required to perform the distributed frequency calculations on each node, merge the results, and then recalculate the cumulative frequency and percentage data. The elapsed time is then written to standard output.

For the experiment, we utilized a dataset containing approximately 10 million records of 100 variables each. With this dataset, a frequency analysis across all records (no filters) performed on a single processor approximately 103 seconds for the original CARE implementation, 150 seconds for the synchronous version of the distributed .NET version, and 128 seconds for the asynchronous version of the distributed .NET version.

Three special filters were created for additional testing. Rather than filtering on the values of particular variables within the dataset, these filters were designed to select a specific percentage of the dataset's records. The first filter selects every record in the dataset. The second and third filters select random subsets of the records, with each record having 10% and 1% chances of being included, respectively.

The dataset and filters were then partitioned into chunks of equal size and divided among the processors. Records were assigned to processors in sequential chunks and relied on the randomly generated filters to reduce the chance of clustering. Seven distinct distributed versions of the base dataset were created (dividing it into halves, thirds, fourths, fifths, sixths, sevenths, and eighths) so that the performance of each version could be tested on any number of nodes between one and eight.

4.2 Methodology

The primary series of tests compared the relative performance of the three distributed CARE versions. Each version performed frequency calculations on the first 100 variables of the test dataset using every combination of the three filters and eight possible node counts, for a total of 24 distinct tests for each version. Each of the 24 tests was repeated ten times.

Additional tests were performed that were designed to determine what effect the other factors mentioned above might have on the three versions of the application. Finally, the synchronous version of the .NET implementation was tested to examine any impact the network configuration might have on performance. Recall that this version supports any style of tree network, as opposed to the other two versions that are limited to star networks. This test compared an eight-node star configuration with a balanced binary tree of depth three. As with the other tests, all three filters were tested, and each test was repeated ten times.

5 PERFORMANCE ANALYSIS

In this section, we evaluate each of the three versions of the application. Performance results and an analysis of this performance are provided for each version.

5.1 Version 1 – Distributed Version of Original CARE

Version 1 of the application (the distributed version of the original CARE application) yielded results that were significantly below the desired performance increase. This is illustrated in Figures 7 and 8, with Figure 7 showing the average running time and Figure 8 comparing the average speed (in records per second). The comparison in Figure 7 compares actual running time to the expected running time (which assumes a linear increase as nodes are added to the system). The comparison in Figure 8 compares the actual speed with the expected speed as the number of nodes increases.

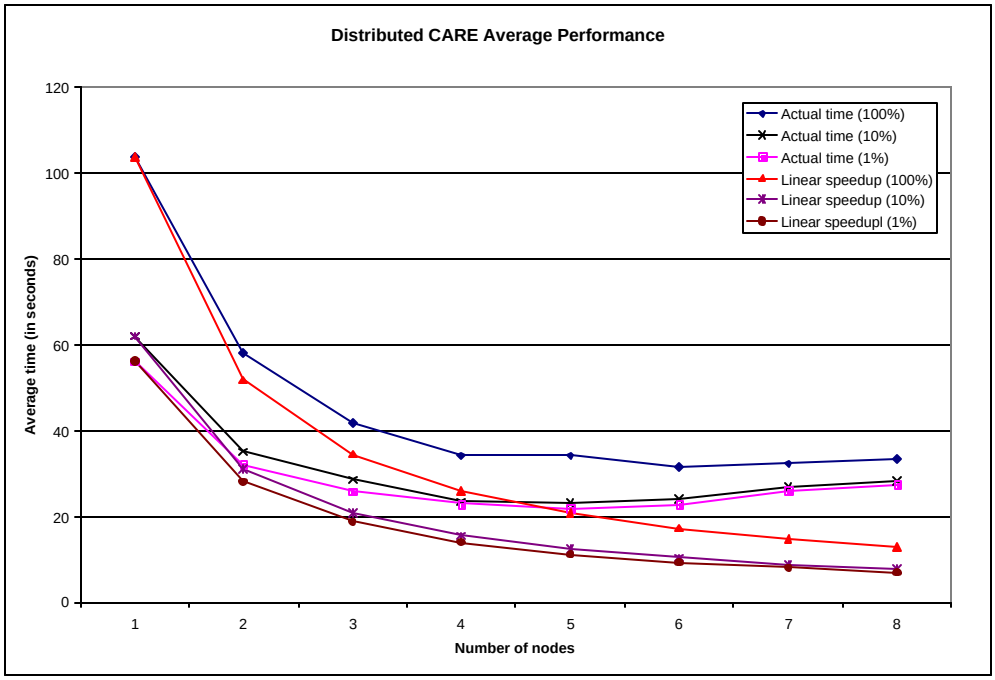


Figure 7: Average Performance of Version 1 (Distributed Version of Original CARE)

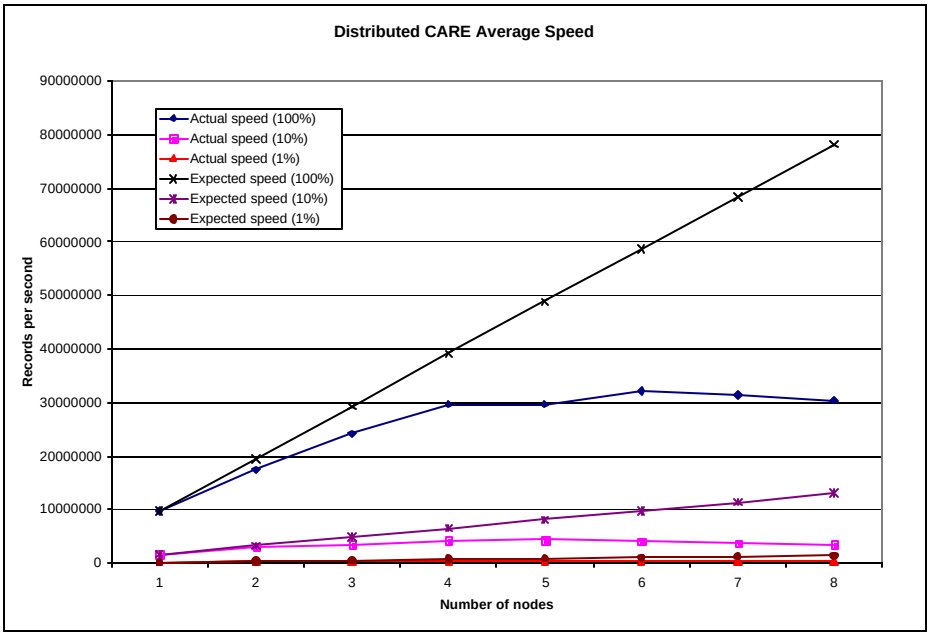


Figure 8: Average Speed of Version 1 (Distributed Version of Original CARE)

5.2 Version 2 - .NET Synchronous Version

As mentioned in the methodology section, two sets of tests were made using Version 2, the synchronous version of the distributed .NET implementation. The first set used a star configuration and the second set used a balanced binary tree configuration. We first examine the results found when using a star configuration.

The results for the star configuration were not as good as desired. In fact, the overall times for Version 2 in this star configuration are higher than for Version 1. We believe that this may be due to overhead caused by extra network configuration information that is sent with the requests. The difference could also be attributed to the inherent differences in performance between the technologies used to implement the two versions. However, some improvements are noted, as Version 2 continues to improve (albeit slowly) after adding at least four nodes to the system. It should be noted that Version 2 never produces times worse than three times longer than the expected times. Furthermore, the choice of filter does not appear to affect the results once three or more nodes are used. Figures 9 and 10 indicate the average running time and speed, respectively, for Version 2.

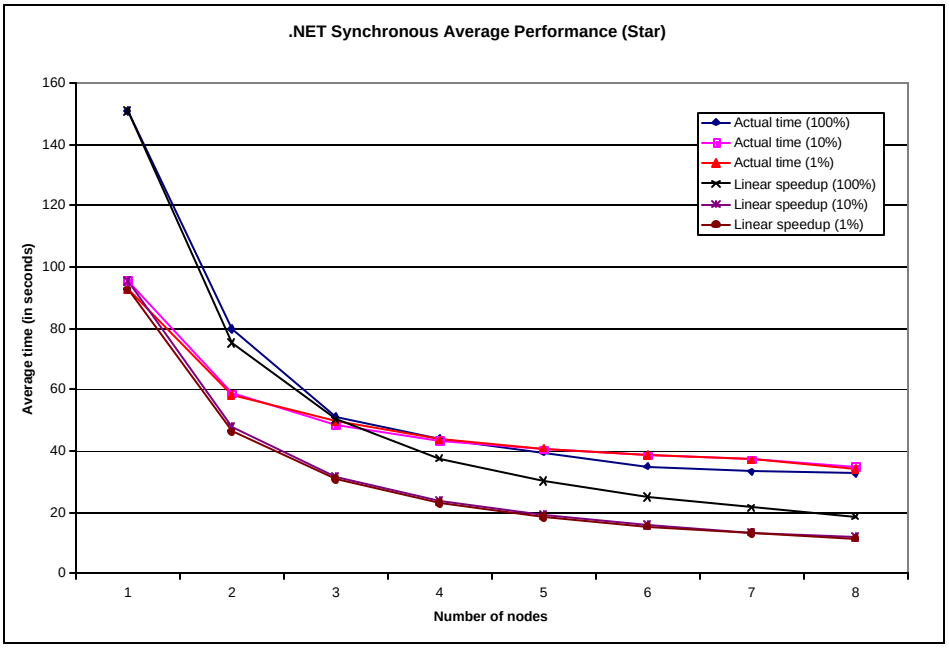


Figure 9: Average Performance for Version 2 (Distributed .NET synchronous – Star Configuration)

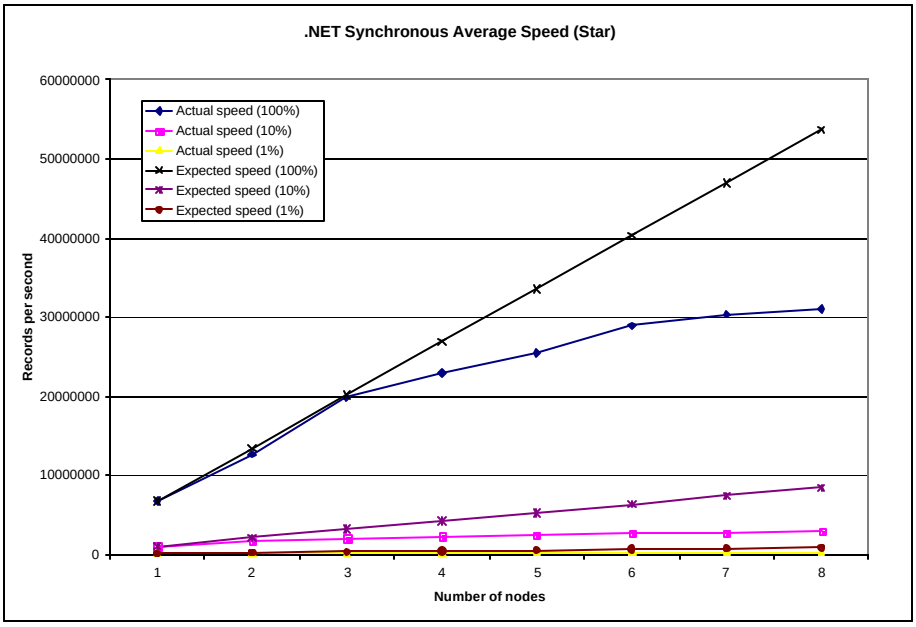
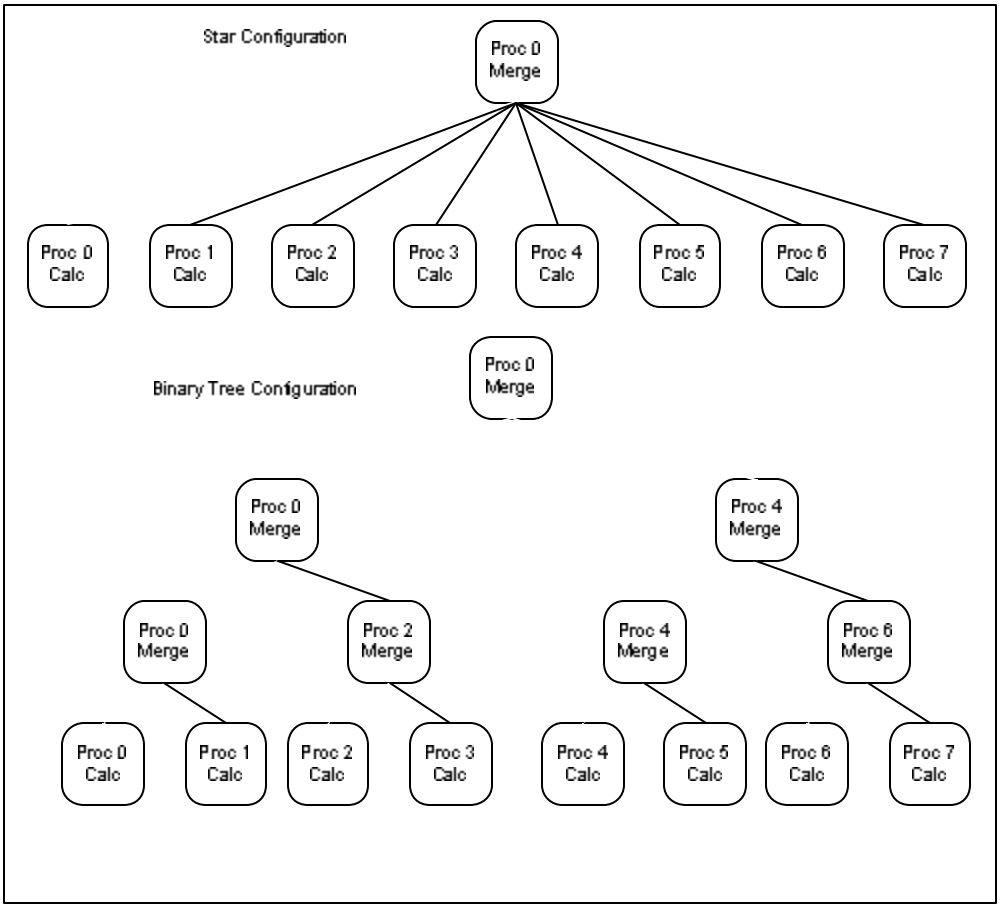


Figure 10: Average Speed for Version 2 (Distributed .NET synchronous – Star Configuration)

In an attempt to improve performance, we also tested Version 2 in a binary tree configuration. In this configuration, multiple merging steps are used to assemble the final frequency data. This configuration reduces the chance that multiple nodes will communicate with a particular node simultaneously. If such collisions occur frequently in a star network, then this configuration might improve performance. On the other hand, if collisions were rare or insignificant, the added complexity could increase the time required to perform a distributed frequency calculation. To reduce network traffic, the tree was configured such that the left child of any merge node in the tree always refers to a merge or calculation component running on the same processor. As a result, the binary tree configuration requires exactly the same number (i.e., seven) of actual connections to a remote server as the star configuration. Figure 11 illustrates the two configurations.



It turns out that the binary tree configuration is even less efficient than the star configuration. These results indicate that network configuration appears to have very little impact

on the performance of the system, as the average times for both configurations differ by less than a second. Figure 12 contains the results of testing the two network configurations.

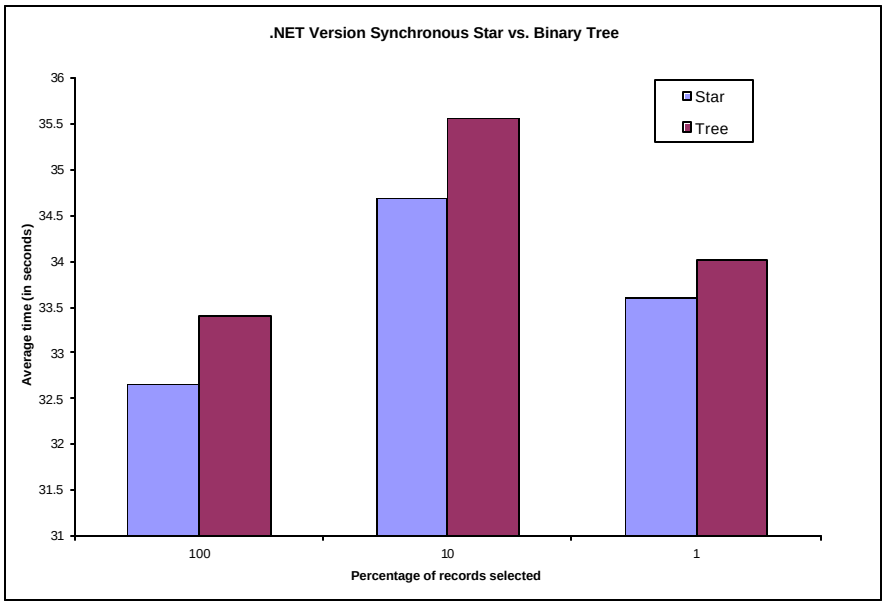


Figure 12: Average performance of Version 2 in a star and balanced binary tree configuration

5.3 Version 3 - .NET Asynchronous Version

When we evaluated Version 3 (the .NET asynchronous version), we tested the hypothesis that the poor performance shown by Versions 1 and 2 was due to the fact that calculations were requested synchronously, one variable at a time. That is, the client application would request a frequency calculation on one variable, wait for the frequency results from each node, merge and recalculate them, and then repeat the process with the next variable.

In Version 3, the client sends a list of variables in a single request. The calculation servers then send back messages containing frequency data back every time they complete calculations for a variable. The client merges these results as they arrive. Once all the servers have responded, the percentage data is recalculated.

Version 3 yielded performance results that were far closer to the ideal. Using the 10% and 1% filters, the actual average times remain almost identical to the values expected for a linear speed increase. The 100% filter results demonstrate an even more impressive result; distributing the calculations provided performance increases that exceed those expected by a linear speed increase. Figures 13 and 14 illustrate the results of the performance analysis for Version 3.

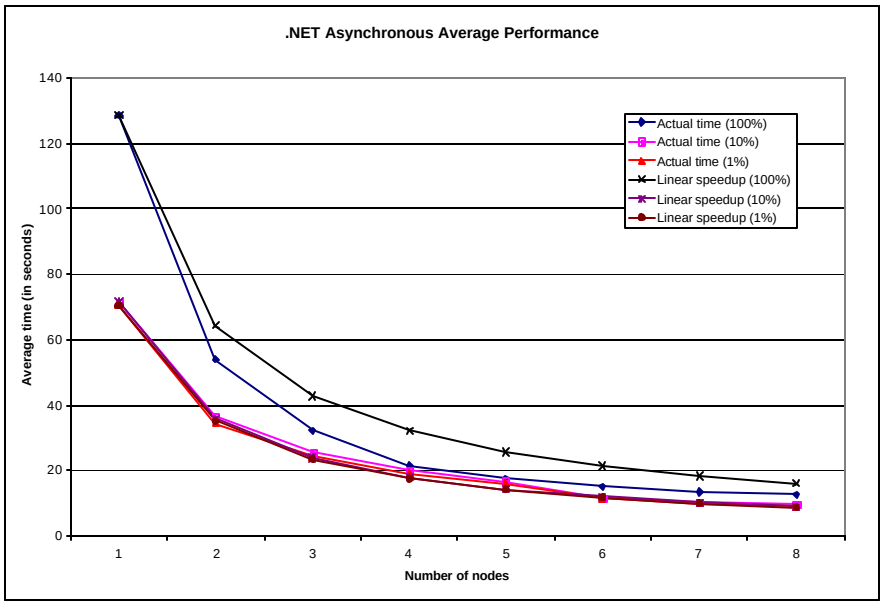


Figure 13: Average Performance for Version 3 (Distributed .NET Asynchronous)

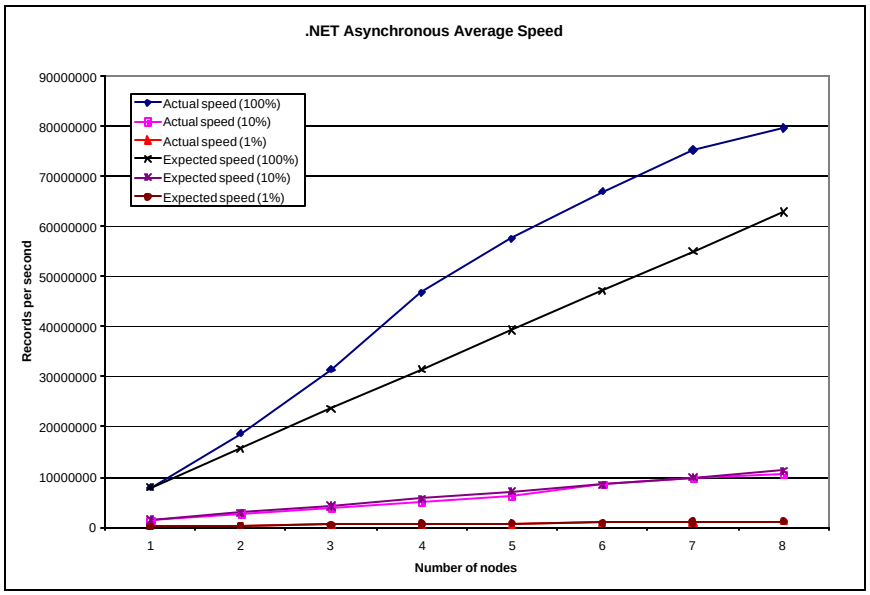


Figure 14: Average Speed for Version 3 (Distributed .NET Asynchronous)

The vast increase in the performance of the asynchronous system relative to the other two versions demonstrates that the primary limiting factor in those versions is the synchronous execution model. By allowing the client to send all its calculation requests in a single batch rather than requesting them one at time, a significant amount of unnecessary idle time was removed. This enhancement allowed the system to reach our expected linear speed increases. The most likely cause of the super-linear speed increase seen in the 100% selection case is the improved cache hit rate realized by reducing the amount of data each node must examine (more of the data remains in main memory).

6 CONCLUSION

This study reinforces the belief that parallel computing using COTS hardware components is an effective method of increasing the performance of data intensive applications. We demonstrated a linear speed increase in our application using inexpensive off-the-shelf hardware components and a relatively small amount of development time and effort. Cluster computing provides the opportunity to go beyond the limitations of currently available hardware without having to pay the high costs usually associated with specialized proprietary technology.

When looking at our results, we still believe that, despite its poor performance results, the middleman technique used to develop the initial distributed application is a powerful tool for parallelizing an existing single-processor system. This technique allows the reuse of almost all of the existing code, substantially reducing system development time. Had the application been designed with distributed operation in mind, the results likely would have been on par with those realized by the asynchronous version.

As a secondary observation, the plug-in system of the .NET implementation provides an effective means of producing distributed versions of an application. While the plug-in technique requires more development time than the middleman approach, this technique allows for the reuse of most of the existing system components. Furthermore, the plug-in system provides more flexibility than the middleman technique, permitting changes in the architecture required to reach the expected level of performance. This approach allows single-processor and distributed operations to coexist within a single application. Additionally, the plug-in approach allowed the .NET implementations to maintain the original user interface component.

7 REFERENCES

1. Y. Deng and A. Korobka, "The performance of a supercomputer built with commodity components." Parallel Computing, 27, 2001, pp. 91-108.

2. P. Merkey, "Beowulf History." http://www.beowulf.org/beowulf/history.html, June 2003.

3. P. O'Neil and E. O'Neil, Database – Principles, Programming, and Performance Second Edition. Morgan Kaufmann Publishers, 2001.

4. Parrish, B. Dixon, D. Cordes, S. Vrbsky, and D. Brown. "CARE: an automobile crash data analysis tool." IEEE Computer Magazine, June, 2003, pp. 22-30.

5. Parrish, S. Vrbsky, B. Dixon, and W. Ni, "Optimizing disk storage to support statistical analysis operations." Forthcoming.

6. TOP500.Org, "Top 500 Supercomputer Sites." http://www.top500.org/dlist/2002/11/#, June 2003.

7. M. Warren, D. Becker, M. Goda, J. Salmon, and T. Sterling, "Parallel Supercomputing with Commodity Components." Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), 1997, pp. 1372-1381.